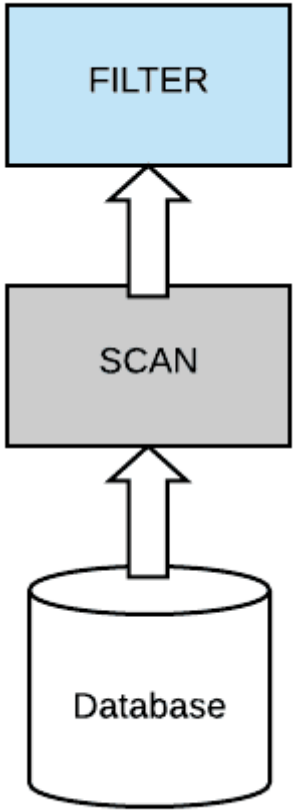
Spark 3.0 为我们带来了许多令人期待的特性。动态分区裁剪（dynamic partition pruning）就是其中之一。本文将通过图文的形式来带大家理解什么是动态分区裁剪。

Spark 中的静态分区裁剪

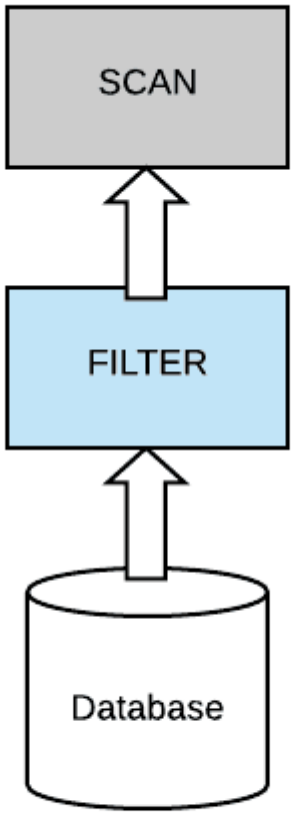
在介绍动态分区裁剪之前，有必要对 Spark 中的静态分区裁剪进行介绍。在标准数据库术语中，裁剪意味着优化器将避免读取不包含我们正在查找的数据的文件。例如我们有以下的查询 SQL：

```
Select * from iteblog.Students where subject = 'English';
```

在这个简单的查询中，我们试图匹配和识别 Students 表中 subject = English 的记录。比较愚蠢的做法是先把数据全部 scan 出来，然后再使用 subject = 'English' 去过滤。如下图所示：



比较好的实现是查询优化器将过滤器下推到数据源，以便能够避免扫描整个数据集，Spark 就是这么来做的，如下图所示：



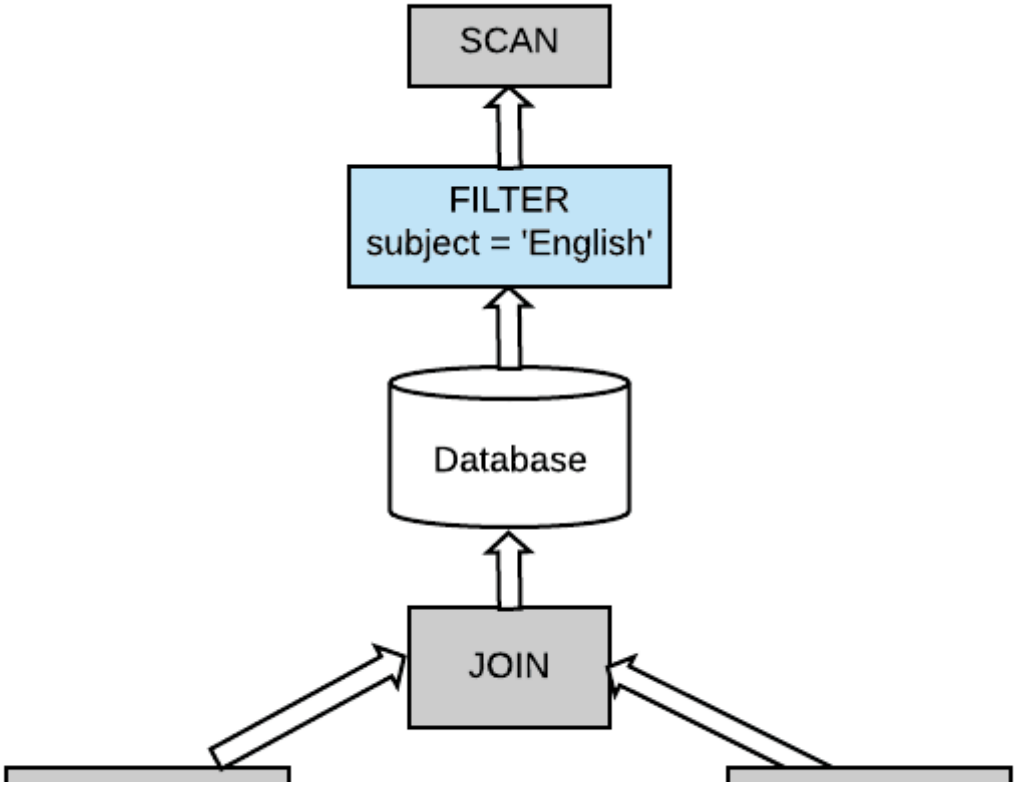
Filter Push Down

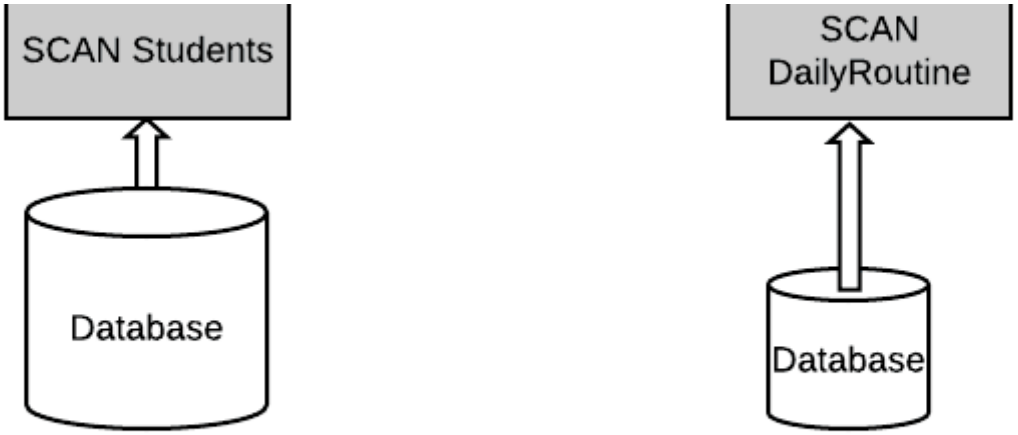
在静态分区裁剪技术中，我们的表首先是分区的，分区过滤下推的思想和上面的 filter push down 一致。因为在这种情况下，如果我们的查询有一个针对分区列的过滤，那么在实际的查询中可以跳过很多不必要的分区，从而大大减少数据的扫描，减少磁盘I/O，从而提升计算的性能。

然而，在现实中，我们的查询语句不会是这么简单的。通常情况下，我们会有多张维表，小表需要与大的事实表进行 join。因此，在这种情况下，我们不能再应用静态分区裁剪，因为 filter 条件在 join 表的一侧，而对裁剪有用的表在 Join 的另一侧。比如我们有以下的查询语句：

```
Select * from iteblog.Students join iteblog.DailyRoutine
where iteblog.DailyRoutine.subject = 'English';
```

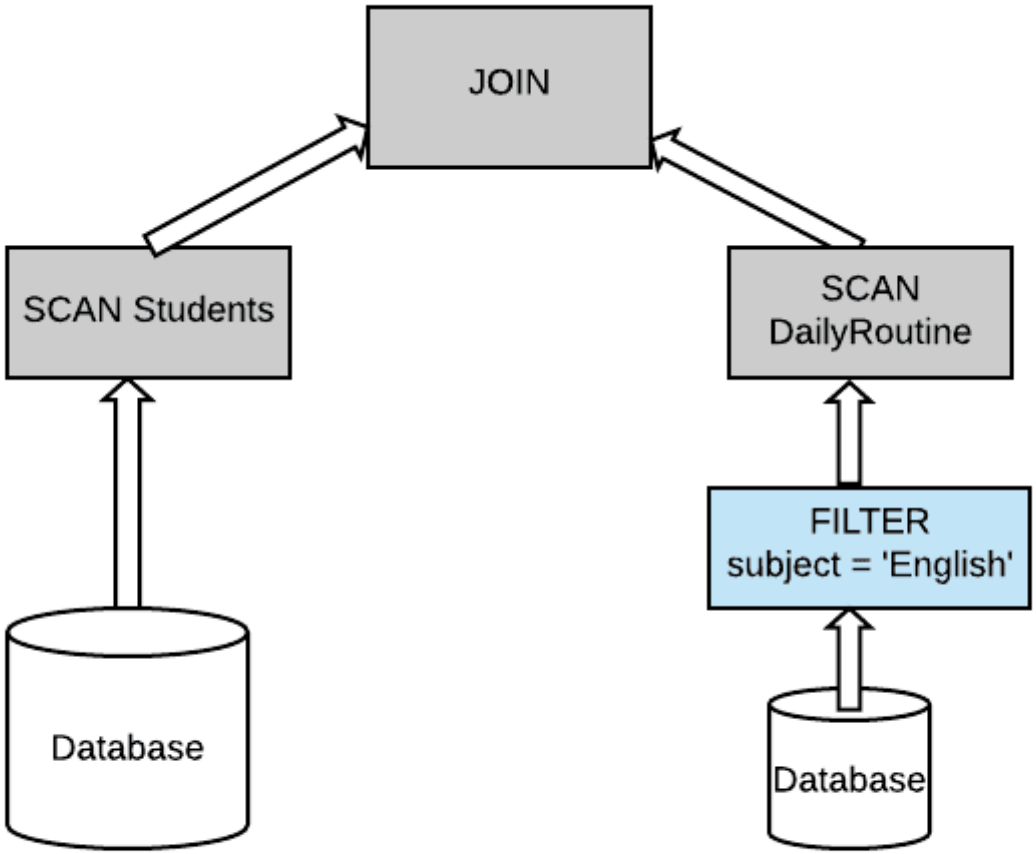
对于上面的查询，比较垃圾的查询引擎最后的执行计划如下：





Simple Workaround for Static Pruning

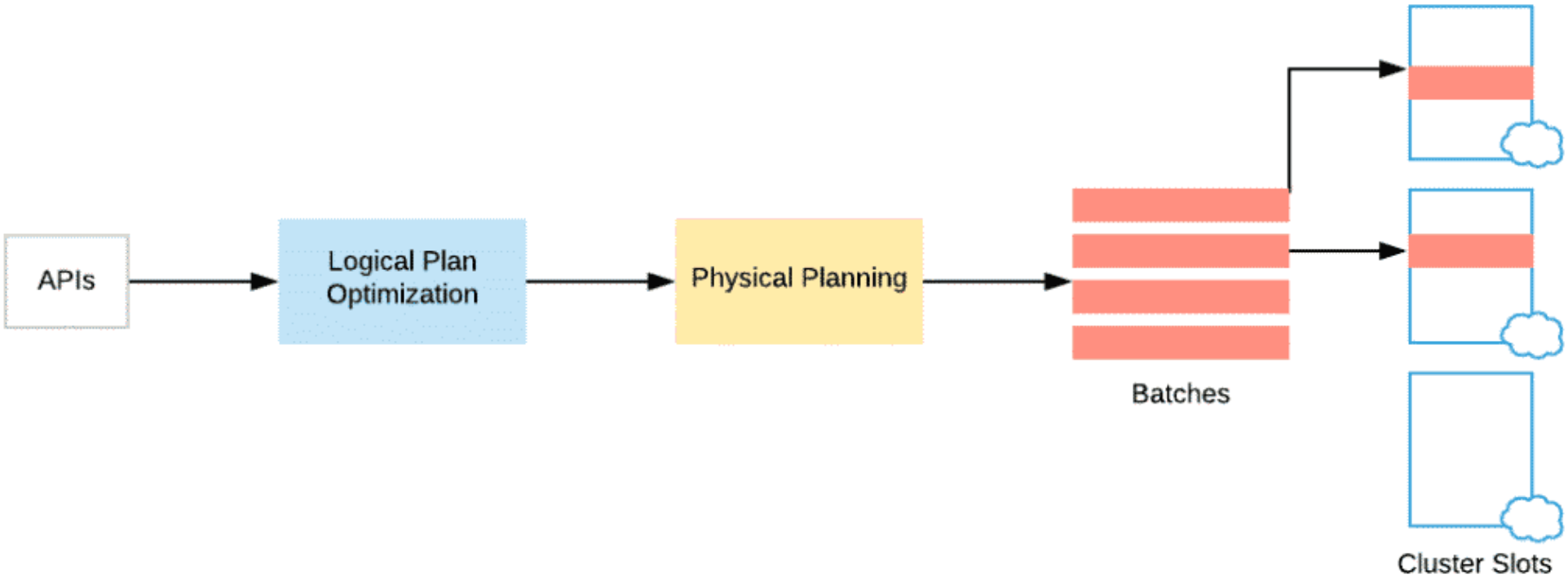
它把两张表的数据进行关联，然后再过滤。在数据量比较大的情况下效率可想而知。一些比较好的计算引擎可以进行一些优化，比如：



其能够在一张表里面先过滤一些无用的数据，再进行 Join，效率自然比前面一种好。但是如果是我们人来弄，其实我们可以把 subject = 'English' 过滤条件下推到 iteblog.Students 表里面，这个正是 Spark 3.0 给我们带来的动态分区裁剪优化。

动态分区裁剪

在 Spark SQL 中，用户通常用他们喜欢的编程语言并选择他们喜欢的 API 来提交查询，这也就是为什么有 DataFrames 和 DataSet。Spark 将这个查询转化为一种易于理解的形式，我们称它为查询的逻辑计划（logical plan）。在此阶段，Spark 通过应用一组基于规则（rule based）的转换（如列修剪、常量折叠、算子下推）来优化逻辑计划。然后，它才会进入查询的实际物理计划（physical planning）。在物理规划阶段 Spark 生成一个可执行的计划（executable plan），该计划将计算分布在集群中。本文我将解释如何在逻辑计划阶段实现动态分区修剪。然后，我们将研究如何在物理计划阶段中进一步优化它。

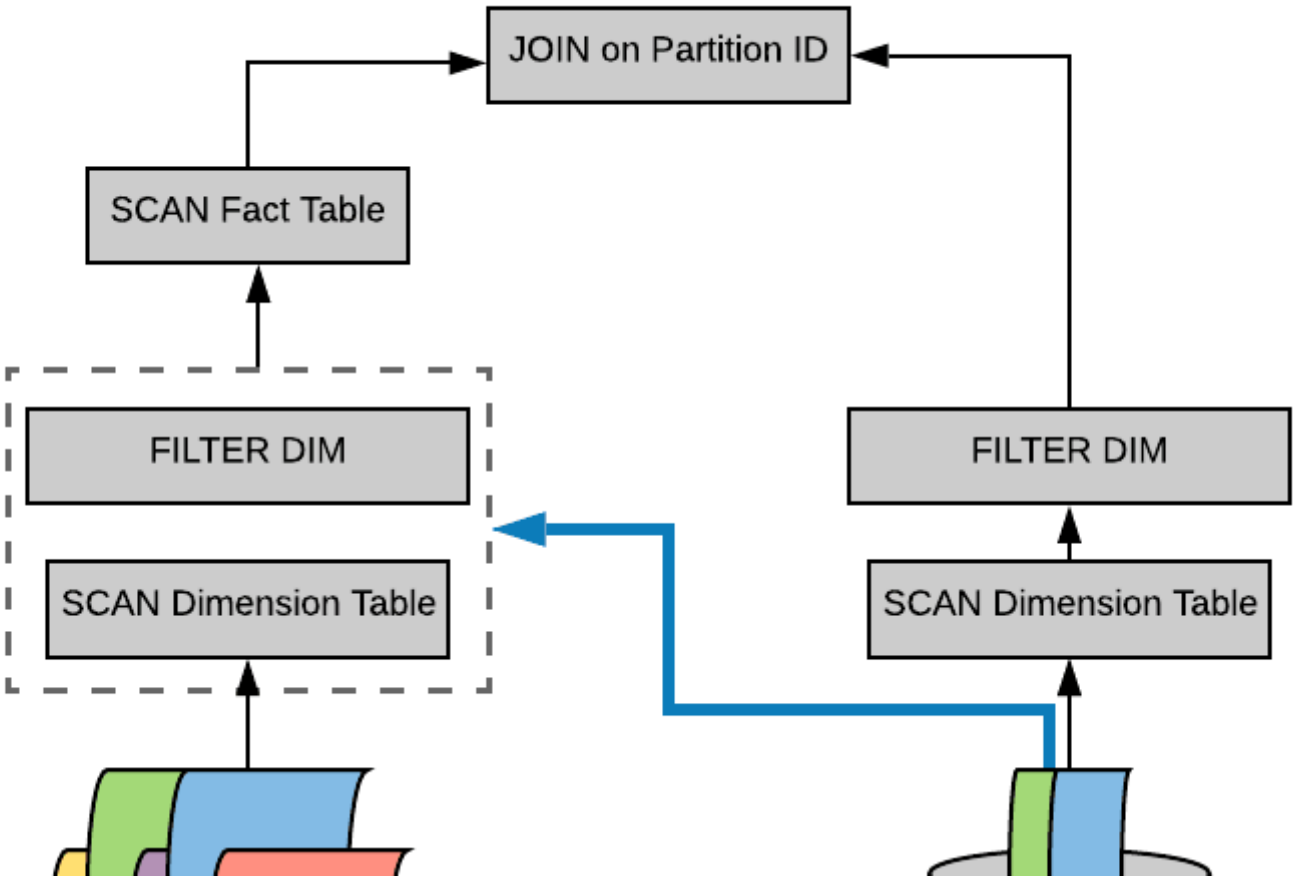


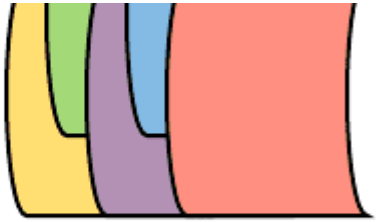
逻辑计划阶段优化

假设我们有一个具有多个分区的事实表（fact table），为了方便说明，我们用不同颜色代表不同的分区。另外，我们还有一个比较小的维度表（dimension table），我们的维度表不是分区表。然后我们在这些数据集上进行典型的扫描操作。在我们的例子里面，假设我们只读取维度表里面的两行数据，而这两行数据其实对于另外一张表的两个分区。所以最后执行 Join 操作时，带有分区的事实表只需要读取两个分区的数据就可以。

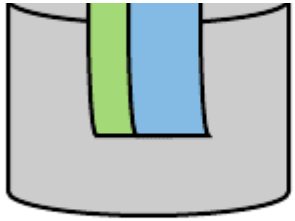
因此，我们不需要实际扫描整个事实表。为了做到这种优化，一种简单的方法是通过维度表构造出一个过滤子查询（比如上面例子为 select subject from iteblog.DailyRoutine where subject = 'English'），然后在扫描事实表之前加上这个过滤子查询。

通过这种方式，我们在逻辑计划阶段就知道事实表需要扫描哪些分区。





Partitioned files with Multi columnar Data

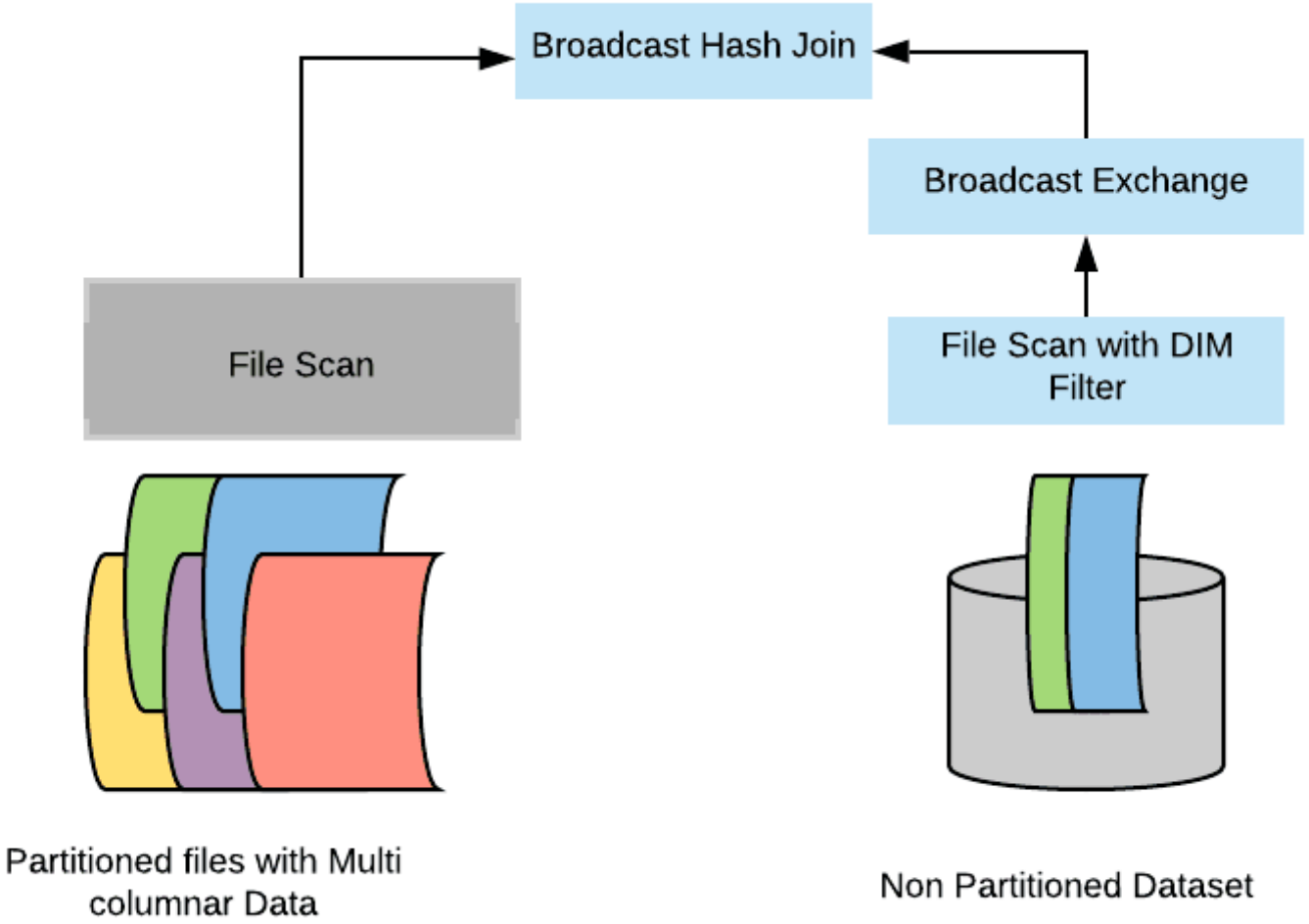


Non Partitioned Dataset

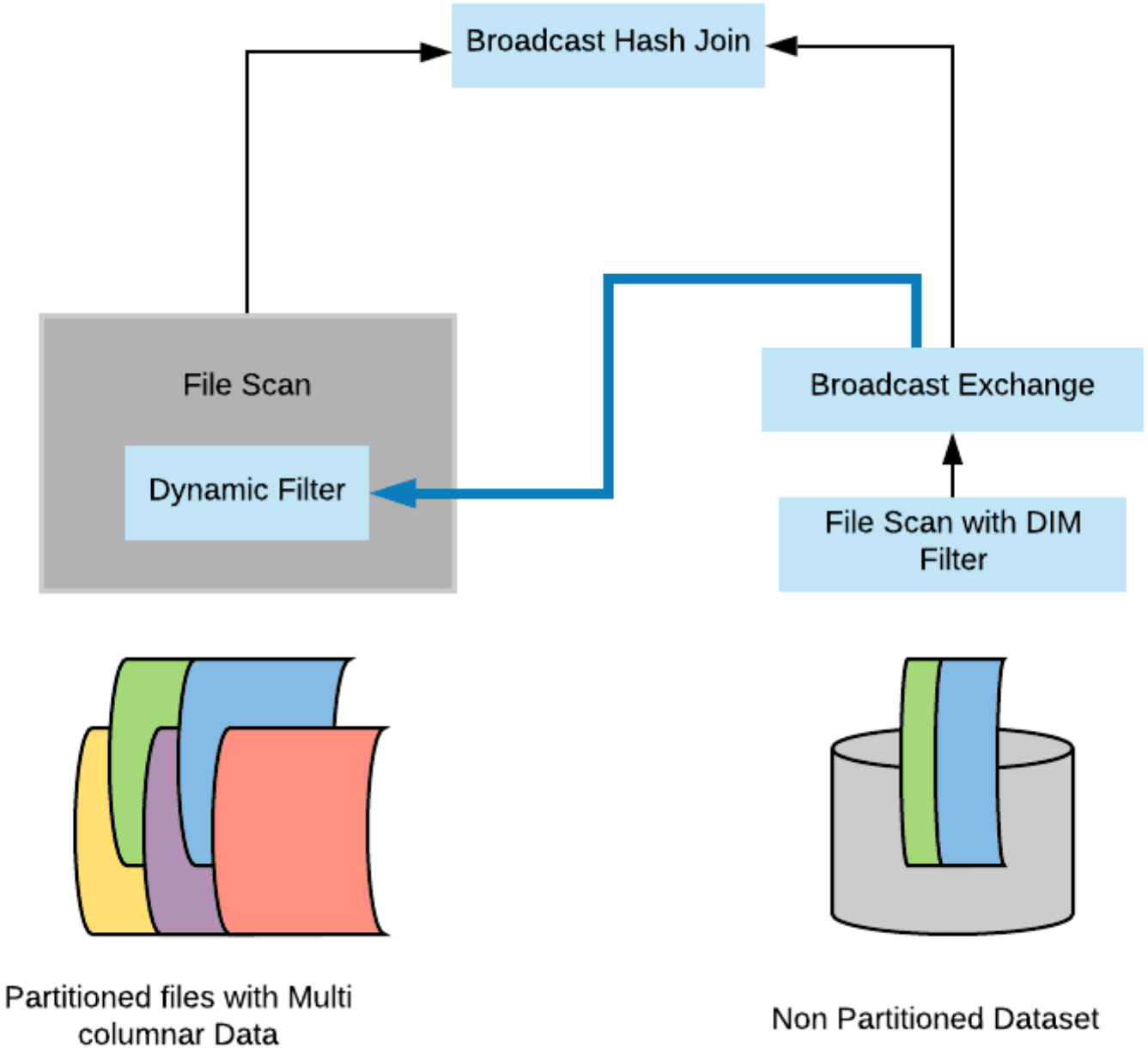
但是，上面的物理计划执行起来还是比较低效。因为里面有重复的子查询，我们需要找出一种方法来消除这个重复的子查询。为了做到这一点，Spark 在物理计划阶段做了一些优化。

物理计划阶段优化

如果维度表很小，那么 Spark 很可能会以 broadcast hash join 的形式执行这个 Join。Broadcast Hash Join 的实现是将小表的数据广播（broadcast）到 Spark 所有的 Executor 端，这个广播过程和我们自己去广播数据没什么区别，先利用 collect 算子将小表的数据从 Executor 端拉到 Driver 端，然后在 Driver 端调用 sparkContext.broadcast 广播到所有 Executor 端；另一方面，大表也会构建 hash table（称为 build relation），之后在 Executor 端这个广播出去的数据会和大表的对应的分区进行 Join 操作，这种 Join 策略避免了 Shuffle 操作。具体如下：



我们已经知道了 broadcast hash join 实现原理。其实动态分区裁剪优化就是在 broadcast hash join 中大表进行 build relation 的时候拿到维度表的广播结果（broadcast results），然后在 build relation 的时候（Scan 前）进行动态过滤，从而达到避免扫描无用的数据效果。具体如下：


Broadcast Hash Join
File Scan
Dynamic Filter
Broadcast Exchange
File Scan with DIM Filter
Partitioned files with Multi columnar Data
Non Partitioned Dataset

好了，以上就是动态分区裁剪在逻辑计划和物理计划的优化。

动态分区裁剪适用条件

并不是什么查询都会启用动态裁剪优化的，必须满足以下几个条件：

•spark.sql.optimizer.dynamicPartitionPruning.enabled 参数必须设置为 true，不过这个值默认就是启用的；•需要裁减的表必须是分区表，而且分区字段必须在 join 的 on 条件里面；•Join 类型必须是 INNER, LEFT SEMI （左表是分区表）, LEFT OUTER （右表是分区表）, or RIGHT OUTER （左表是分区表）。•满足上面的条件也不一定会触发动态分区裁减，还必须满足 spark.sql.optimizer.dynamicPartitionPruning.useStats 和 spark.sql.optimizer.dynamicPartitionPruning.fallbackFilterRatio 两个参数综合评估出一个进行动态分区裁减是否有益的值，满足了才会进行动态分区裁减。评估函数实现请参见 org.apache.spark.sql.dynamicpruning.PartitionPruning#pruningHasBenefit。
本文主要翻译自：https://blog.knoldus.com/dynamic-partition-pruning-in-spark-3-0/